Interactive Python Shell for jAliEn

Ådne Garstad Nerheim

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

20 November 2019

Acknowledgements

I would like to thank all those who helped me through the course of this study and contributed to this project. First I would like to thank my supervisors Bjarte Kileng, Håvard Helstrup and Kristin Fanebust Hetland for their feedback and guidance during the writing of this thesis and development of the prototype. Additionally I would like to thank the jAliEn staff for their help and feedback on the prototype. I would like to thank Maksim Melnik Storetvedt for giving me valuable insight into the inner workings of jAliEn and his help on defining the scope of the project. Last, I want to thank my family for their support, understanding and encouragement during this project and my education as a whole.

Ådne Garstad Nerheim

Bergen, November 2019

Abstract

Computing Grids are collections of interconnected computing resources used to solve large computational problems. These resources are used by the ALICE (A Large Ion Collider Experiment) Collaboration, ALICE being one of the four main experiments at the LHC (Large Hadron Collider) at CERN, for the transfer, storage and analysis of experimental data. Grids are managed by Grid middleware. jAliEn is the grid middleware developed by the ALICE Collaboration and is in the process of replacing the old grid middleware AliEn (ALICE ENvironment).

This thesis investigates how an interactive CLI (Command Line Interface) written in python which communicates with the jAliEn central services can be created and whether or not the resulting solution is a viable alternative/replacement of the old client(jShell). For this purpose, criteria were outlined to determine what constitutes an interactive CLI, the resulting prototype was then compared to jShell to determine if the solution could be considered viable.

List of Figures

List of Tables

Chapter 1

Introduction and Background

1.1 Background

A Large Ion Collider Experiment (ALICE) is a heavy-ion detector on the Large Hadron Collider (LHC) at CERN. ALICE is dedicated to the study of heavy ion collisions and recorded its first collision in 2010[1]. During operation the detectors gather an immense amount of data which has to be transferred, processed and stored. To manage that, you need computing power and technology able to handle the huge requirements. This is the job of the Grid.

Ian Foster, a prominent figure in the world of Grid computing defined the Grid as a system that "*coordinates resources that are not subject to centralized control using standard, open, general purpose protocols and interfaces to deliver nontrivial qualities of service*"[2]. The grid is a collection of heterogeneous data centers located at different locations. When combined, these data centers fulfill the requirements needed.

To connect these resources, and make them act as one entity to the end users you need to use a grid middleware. The grid middleware is responsible for security, transfer of data, storing of data and much more. This will be discussed in greater detail later.

The grid middleware used by the ALICE experiment at CERN is the Alice Environment (AliEn). It started development in 2000 and was quickly deployed. JAliEn, which is supposed to replace the existing AliEn services has been in development since 2010[3]. JAliEn consists of several components (discussed in detail later), one of which is the central services. It plays the role of the middleman between the users and the sites. The central services supports WebSockets for establishing a connection.

1.2 Motivation for this project

The client currently used by members of the ALICE collaboration (jShell) to interact with the central services is written in Java. As jAliEn is now beginning to be implemented and replacing AliEn there is a need for a client written in Python. Python is open-source and all versions of Python 3 are backwards compatible (unlike Java which is not) and is therefore preferred over Java.

The jAliEn central services WebSocket connection expects a command as input and if accepted will return the response to this command in the form of a json[1] file. Ideally you should be able to interact with the grid in a manner resembling the use of a terminal locally on you computer, retaining as many features as possible. This is the job of the interactive Python shell for jAliEn.

1.3 Goals

The objective of this thesis is to explore the viability of an interactive Command Line Interface (CLI) written in python which connects to the jAliEn central services using websockets and look at how it compares to the existing solution. In short, this thesis aims to answer these two research questions:

How can an interactive shell which meets the given criteria and connects to jAliEn via websockets be created?

How does this solution compare to the current implementation, and is it a suitable replacement or alternative?

1.3.1 Interactive Command Line Interface

A CLI is a simple interface on the screen where you can enter commands and have the results displayed back to you. The actual processing of the commands is done by the central services, while this application handles the interactivity (see figure 1).

[1] JSON (JavaScript Object Notation) is a lightweight data-interchange format which can be easily read and written by humans.

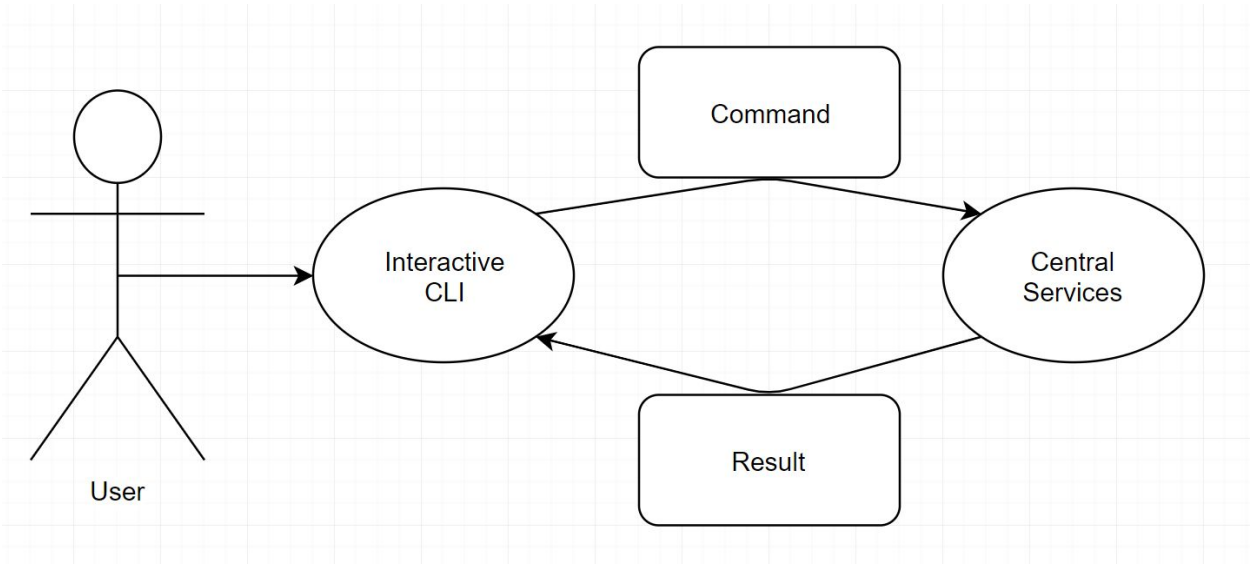

Figure 1.1: Interactive CLI for jAliEn illustration

This section will present some of the key features of an interactive CLI as well as the criteria for creating a viable solution. For an explanation on how they were implemented, see chapter 5.

- **Tab Completion** - Also called auto-completion, this feature allows you to enter the first few characters in a command or file name and press the “tab” key to auto-complete the string. If there is more than one possible completion, all possible completions will be displayed instead.
- **Scripting** - Allows a user to make their own scripts with the available commands.
- **Robustness** - Since this application relies on a websocket connection it should handle cases where the connection closes for a short time. It should also catch and handle invalid inputs before they are sent.
- **Light** - A CLI contains no graphical assets and should have few dependencies, as a result the file itself should be small and easy to install.
- **Maintainability** - The code should be easy to read and adding additional commands (including ones which uses other commands) should be easy to implement.
- **User friendly** - All the features above combine to create a much more user friendly interface when compared to a CLI without interactivity.

1.4 Method

A prototype application will be made to best answer the research questions and present a solution. The prototype will be an interactive CLI capable of communication with JAliEn and hopefully implement all features/criteria discussed in section 1.3.1.

1.5 WebSockets

WebSocket is a protocol which allows for a full-duplex TCP connections[4]. HTTP requires two open connections for two-way communication, WebSocket requires only one. The WebSocket protocol enables communication between a client and a webserver using less overhead than other alternatives allowing for real time data transfer. When using the WebSocket protocol, the server can send content to the client without the client first requesting it, keeping the connection open and allowing messages to be sent back and forth(see figure 1.2 and 1.3). In essence, traditional HTTP communication is like a set of walkie-talkies where only one part of the conversation can talk at any given time whereas the WebSocket protocol acts like cellphones, allowing both to communicate simultaneously.

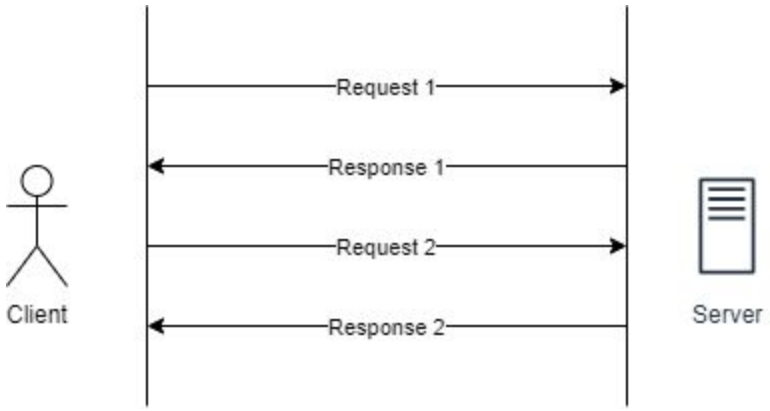

Figure 1.2: HTTP Communication Example

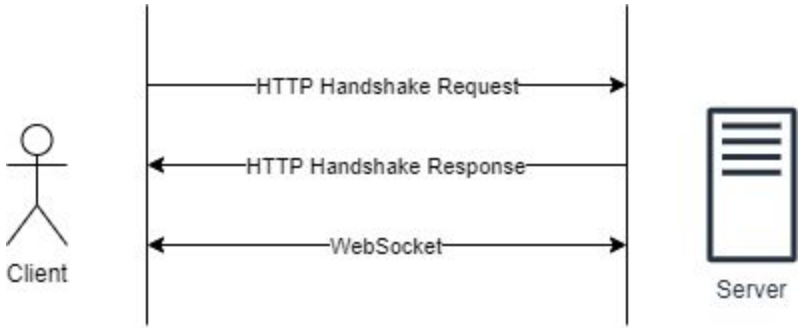

Figure 1.3: WebSocket Communication Example.

The protocol was designed to work with existing web infrastructure. As a result it uses the same port as HTTP (80) and HTTPS (443) and to ensure backwards compatibility; a WebSocket connection starts out as an HTTP connection. What is referred to as a “WebSocket handshake” is in fact the protocol switch from HTTP to WebSocket.

When a client wishes to establish a WebSocket connection, it sends a WebSocket handshake request and the server returns a WebSocket handshake response (see table 1.1 and 1.2)[34]. Once the WebSocket connection has been established, both client and server can send and receive messages using full-duplex communication.

```
   GET /chat HTTP/1.1
   Host: server.example.com
   Upgrade: websocket
   Connection: Upgrade
   Sec-WebSocket-Key: dGhlIHNhbXsZSBub25jZQ==
Origin: http://example.com
   Sec-WebSocket-Protocol: chat, superchat
   Sec-WebSocket-Version: 13
```

Table 1.1: Websocket handshake request from client[34]

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Table 1.2: WebSocket handshake response from server[34]

Chapter 2

History

2.1 CERN and ALICE

The idea for a world-class research facility in Europe can be traced back to the 1940s.[5] CERN (Conseil Européen pour la Recherche Nucléaire) was given its name at a UNESCO convention in 1951 and in June 1953 the final draft of the CERN Convention was agreed upon and signed by 12 new Member States. Over half a century later, our understanding of matter has grown (or more appropriately “shrunk”) and CERNs main area of research is now particle physics. The laboratory has also grown to include 23 member states[6]. Now more than 17 500 Scientists of 110 nationalities from institutes in more than 70 countries are active at CERN[7].

Worldwide, CERN is perhaps best known for the LHC (Large Hadron Collider) the world's largest and most powerful particle accelerator[8]. The LHC was first started on september 10th 2008 and contains two high-energy particle beams traveling in opposite directions at close to the speed of light before they are made to collide. There are 4 places along the ring where these beams cross. Each intersection is home to one of 4 experiments: ATLAS (A Toroidal LHC ApparatuS), CMS (Compact Muon Solenoid), LHCb (Large Hadron Collider beauty) and ALICE (A Large Ion Collider Experiment).

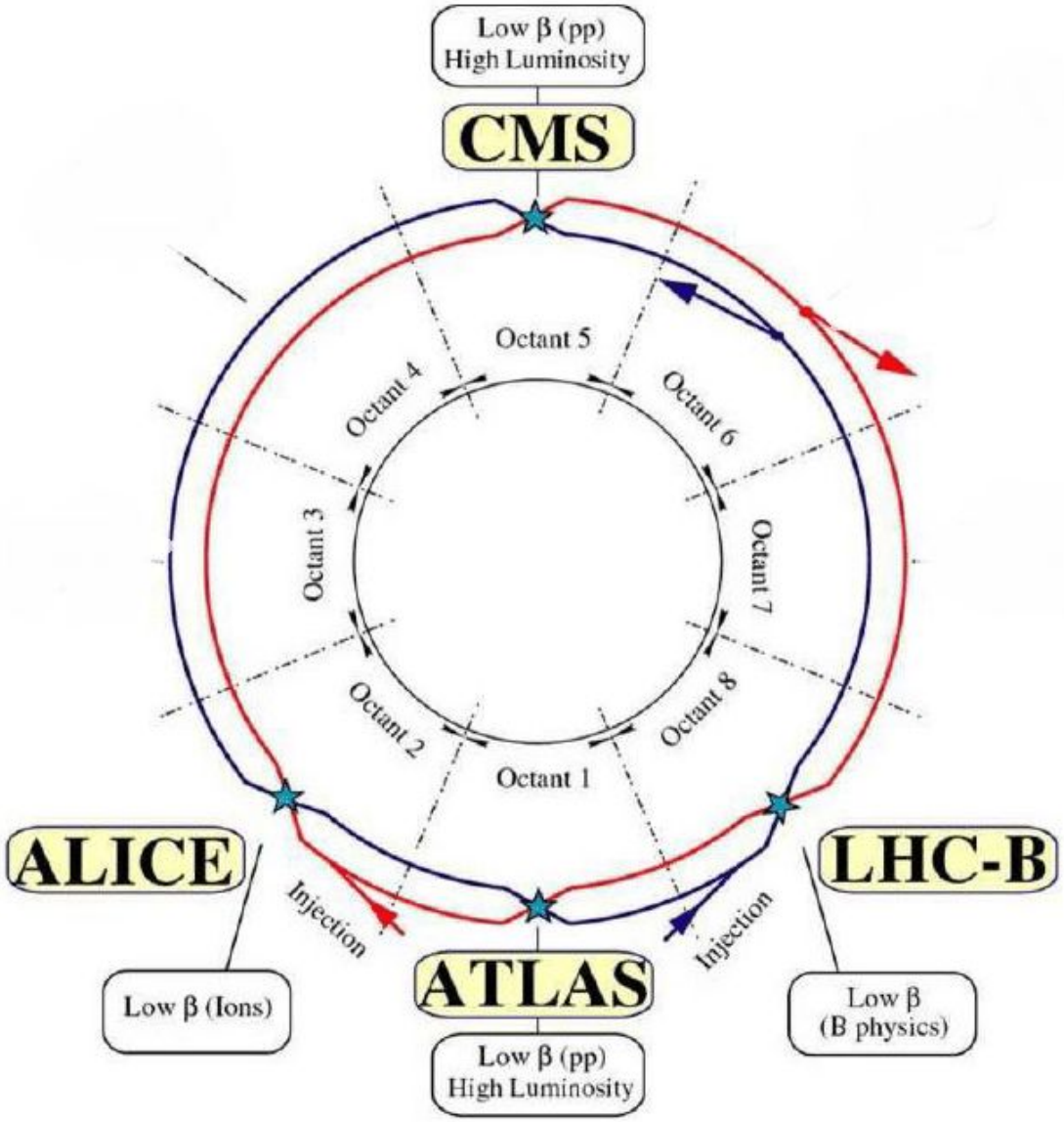

Figure 2.1: Schematic Layout of the LCH[10](Modified)

ALICE is a heavy-ion detector designed to study the physics of strongly interacting matter at extreme energy densities[9]. When the LHC is running and the detectors are collecting data from the collisions, Gigabytes of data is generated every second. That data has to be transferred, processed and stored. The grid middleware is used to organise offline processing of this data. The grid middleware used and developed by ALICE is called AliEn.

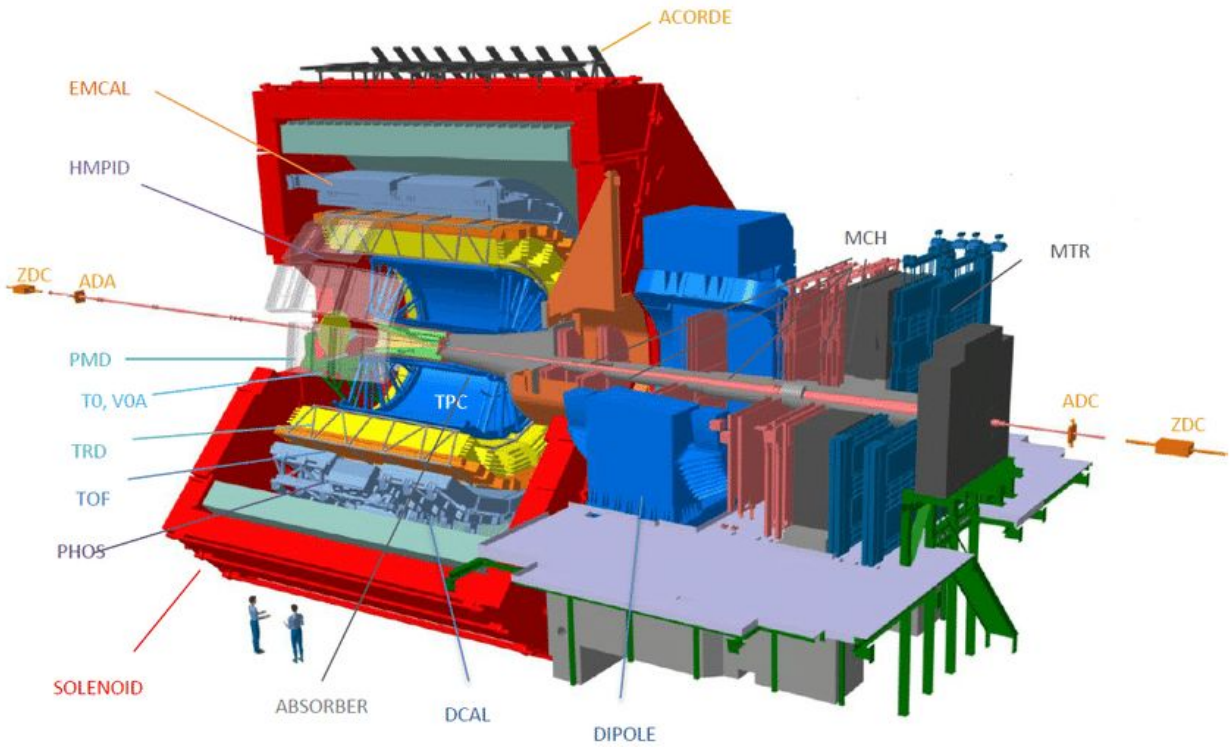

Figure 2.2: The ALICE experiment during LHC Run 2[11].

2.2 Grid Computing

Grid computing was an idea originating from the late 1990's. Researchers wanted a computing infrastructure that would provide computing power on demand, and they referred to it as the Grid[12]. The name Grid computing was actually a metaphor of the power grid. The reasoning behind the name was; it should be as easy to get computing power from the computing grid as it was to get electric power from power grid[13].

In 1995 at the Super Computer conference, the first large-scale Grid was demonstrated. It was called the Information Wide-Area Way[42]. It included 17 geographically distributed computing sites, and demonstrated over 60 applications from various scientific fields. This marked the beginning of the Globus Toolkit. The Globus Toolkit consisting of contains components such as security, job submission and resource management. Components from the Globus Toolkit are used in other Grid middleware projects, including AliEn.

2.3 The Unix Interactive Shell

A shell is a UI for accessing the operating systems' services(see figure 2.3). More commonly known as Bash, the bourne-again shell is a free unix shell written for the GNU project[36]. Bash is the most common shell installed with Linux distributions[37].

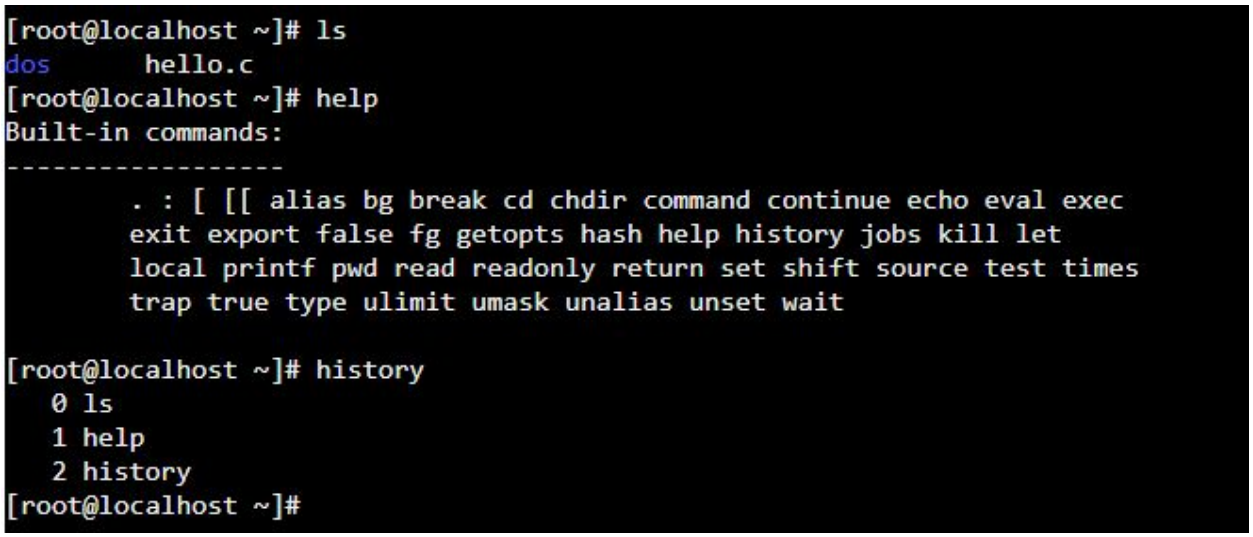

Figure 2.3: Bash shell example

The bash shell contains many useful built in commands and functions. This includes commands like *history* which stores previously entered commands and *help* which returns a list of all available commands when no arguments are given or the description of a command when one is given as an argument. It also includes functionalities like aliases which allows one function to be referred to by more than one command (for example *quit* can have the alias *q*)

When a user presses the tab key, bash uses command line completion. Bash does this by utilizing the readline library. Readline provides the all important line-editing and history capabilities of many CLIs including bash. It is readline, not bash which catches the tab press and then edits the line currently being written. However it is the complete function of bash which provides the suggestion(s).

There are many forms of completion in bash; path, file, user, host and variable completion [41].

- Path-name completion is both the most known and used completion. It allows you to complete executable files.
- File-name completion allows the user to complete file and directory names.
- User-name completion allows the user to complete user names if prefixed by “~”.
- Host-name completion allows the user to complete host names if prefixed by “@”.
- Variable-name completion allows the user to complete variable names if prefixed by “$”.

Knowing the ins and outs of the bash shell can be important when developing a CLI. It contains useful features, and is widely used. Designing a CLI with bash in mind will result in an application which is more user friendly because bash is familiar to the majority of users. As a result, the prototype will attempt to emulate bash wherever possible(see section 5.2).

Chapter 3

Grid Computing

3.1 Introduction

In very simple terms, the grid acts as a service where a user can submit a job and later get the result of that job back. This seemingly simple tool is incredibly important in many areas of modern science. Studies in physics and chemistry like that at CERN, and biomedical research where grid computing is used to further our understanding of the human genome and helps in the development of new drugs. This chapter will introduce some of the key aspects of grid computing.

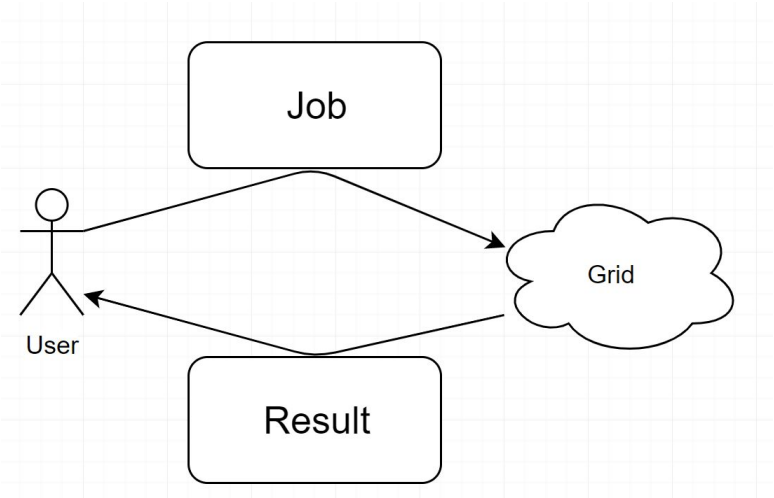

Figure 3.1: Interaction between an end user and a grid.

3.2 Grid Middleware

Security and job distribution are only some of the tasks handled by the grid middleware. In essence, the grid middleware is a bridge between the user and the resources on the grid (see figure 6). Grid middleware should provide:

- *Single sign-on:* Being able to access all available functionalities without having to supply additional account information.

- *Information Services:* Provide information about the resources, including current status, services, files and jobs.
- *APIs and services:* Enable other application to take advantage of the middlewares capabilities.
- *User Interface:* A convenient way for users to interact with the grid.

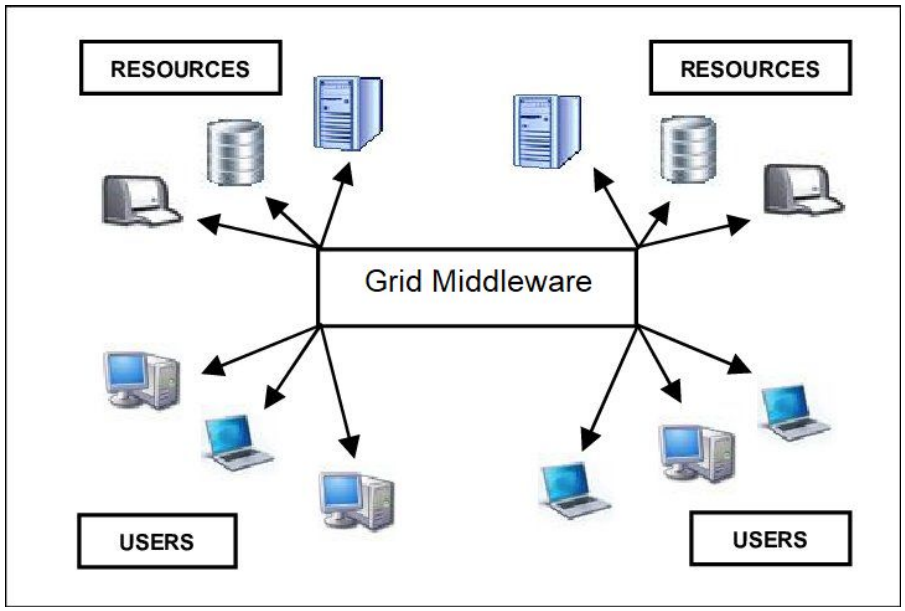

Figure 3.2: A visual representation of a grid middleware[22]

3.3 Security

Providing secure transfer of data is the Grid's responsibility. To ensure security between sites and enforce the distribution of resources it has to establish a chain of trust. Security in grid computing ecompasses four areas:

- *Data integrity*: Ensuring the data sent is the same as the data received
- *Data confidentiality*: Ensuring the data sent can only be read by the intended receiver.
- *Authentication*: Ensuring the identity of a user.
- *Authorization*: Ensuring the provided identity have access to the requested resource

This section will detail how these four areas of security is handled by a Grid.

A commonly used authentication system is login, where every user has a username and password. There are several problems with using such a system. Users tend to reuse their passwords across several services. It doesn't matter if you store your passwords in a secure manner, when the same username and password combination exists in several other insecure databases. If a service enforces strict rules for passwords (at least some number of digits long, forcing a change every few months, not so similar to previous passwords, etc) this fails when the users write it down on a post-it note and sticks it to their monitor.

A grid usually doesn't have more than a few thousand users at most. Since a single compromised user can't cause too much damage the example above isn't the biggest issue. Traditionally, every user has their own public and private key. The public key is readily available, and the private key is kept secret. Any message encrypted by the sender's private key can only be decrypted by their public key, ensuring data integrity. At the same time, any message encrypted by the receivers public key can only be decrypted by the corresponding private key, ensuring data confidentiality(see figure 4).

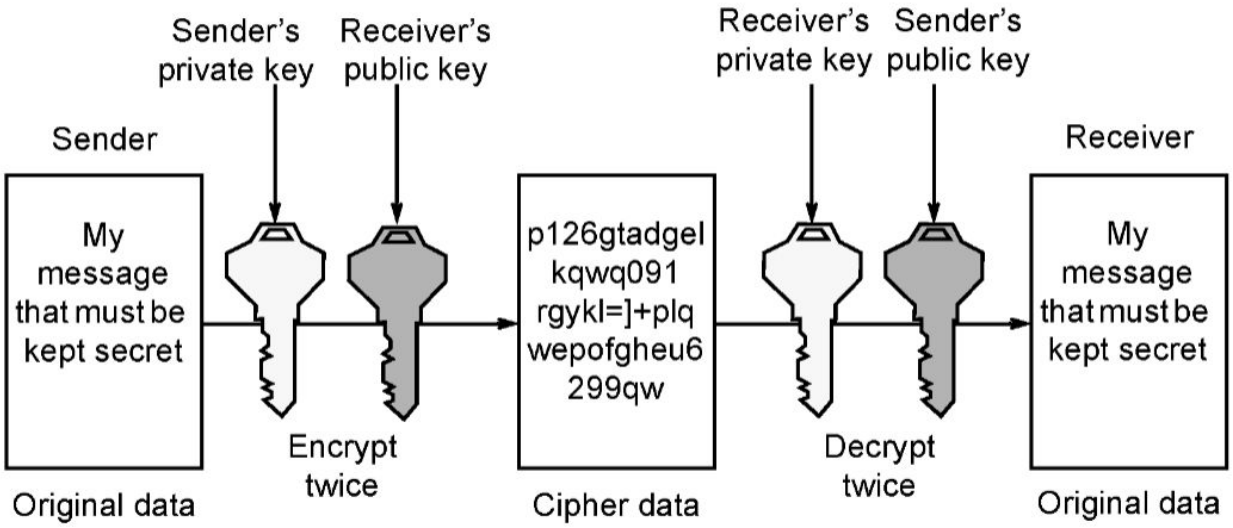

Figure 3.3: PKI Example[42]

There are major issues with this method, firstly the public key must be the one held by the receiver and the private key must be held securely. Secondly, it's slow. The solution to all our problems can be found in the Public Key Infrastructure (PKI). A PKI consists of software and hardware elements that a trusted third party can use to establish the integrity and ownership of a public key[14]. The concept of trust is very important in a grid.. All sites needs to agree to trust one party. This party is called a Certificate Authority (CA). In some cases, some sites will have their own CA but as long as the Root CA trusts the site's CA, trust is still upheld (see figure 5).

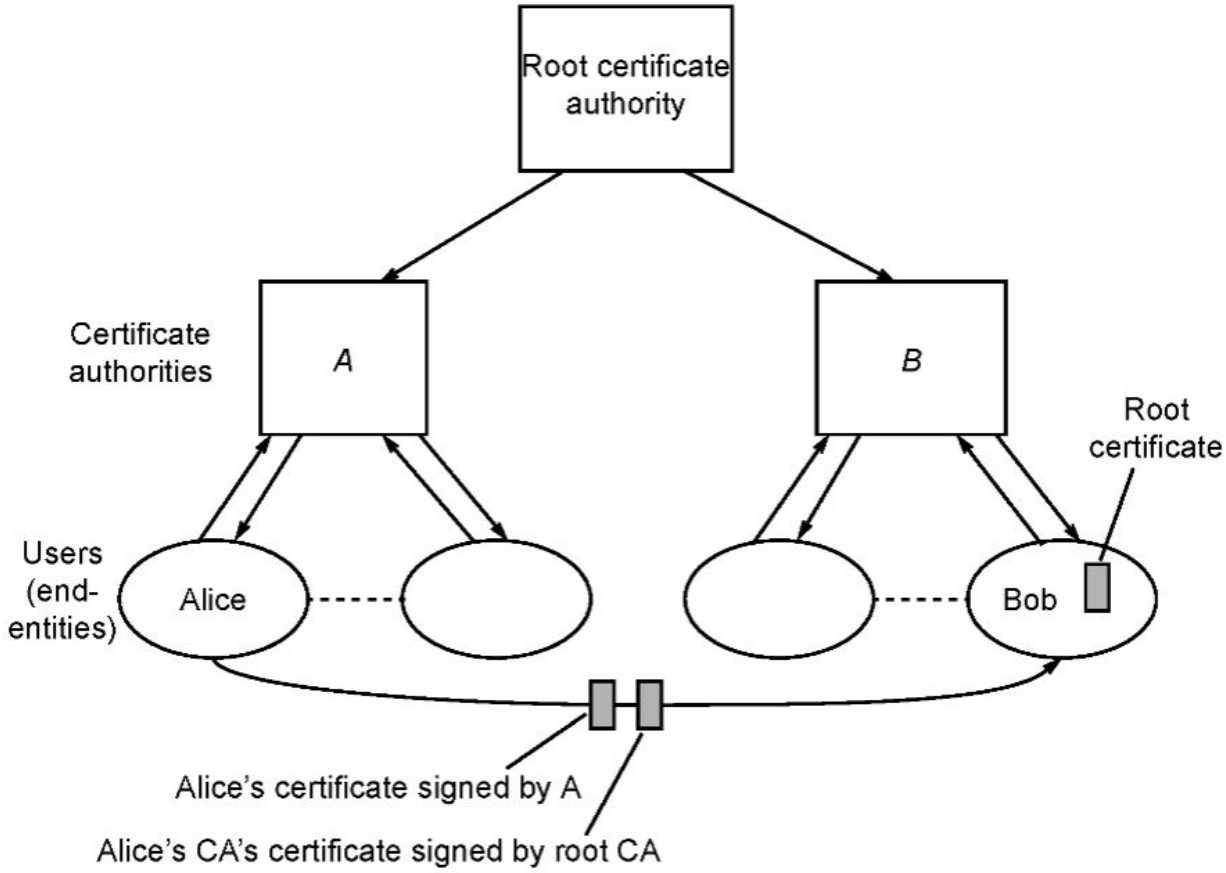

Figure 3.4: CA Example[42]

3.3.1 Certificates

A certificate is put simply; a digital document used to validate that the public key belongs to the user. It contains the name of the certificate holder, their public key and the signature of a trusted CA which certifies that the public key does in fact belong to the user named in the certificate. The CA signs the certificate using their private key, which can be validated by their public key. This way the public key can only be "faked" if someone manages to get ahold of the CA's private key. In the case of a tiered system as seen in figure 5, CAs A and B will have their public key signed by the root CA, who's certificate is signed by itself.

The certificate is used to validate a user by some remote service. However in a grid environment these remote services often have to interact with other remote services to process or transfer files on the user's behalf. In order to act on the user's behalf, the remote service is given a *proxy certificate* signed by the user who created it. The proxy certificate has its own public and private key and for security purposes has a short life cycle of 12 - 48 hours.

Using figure 5 as a base. A proxy certificate created by Bob will be signed and therefore trusted by Bob, who's certificate is signed and trusted by his CA, which in turn is trusted by the root CA establishing the chain of trust which is so crucial in a grid environment.

3.4 Jobs

Grids exists to share computing resources. These resources are needed to process jobs. A job in grid terms, is a file written by the user which is to be executed on the grid. This section will explain what a job is, and how jobs are managed by the grid.

3.4.1 Job Definition

Jobs in a grid setting can be executables, OS commands, scripts, programs that first need compiling and more. The job itself might also have other parameters, for example; which files to use as input, what environment it needs to run on, and which files to output too. The exact description of a job differs from grid to grid. AliEn for instance uses Job Description Language(JDL)[15].

```
Type = "Job";
JobTy pe = "Normal";
Executable = "exec";
StdInput = "input.txt"
StdOutput = "output.txt";
StdError = "error.txt";
```

Table 3.1: JDL Example

Table 1 demonstrated only some of the attributes which can be set in JDL, other attributes such as computing requirements, your email address so you will be notified when the job finishes and TTL(time to live) which dictates how long the job is allowed to run.

3.4.2 Job Distribution

There is never a shortage of jobs that need to be processed. Once a job has been submitted to the grid it then has to be handed over to a site with the appropriate resources as described in the job. There are a lot of things which need to be taken into consideration here. The ultimate goal is to maximize the effectiveness of the grid. This is made particularly challenging as two sites are never the same, and two sites might give different performances on the same job. This can be caused both by the environment of the site and by the files necessary to run the job. If the data needed for a job is stored in Romania, it will take longer to run the job in Ireland than another location in Romania.

Again the specific solution to this problem differs from grid to grid, however one solution involves the sites “advertising” its available resources across the grid, which allows for easy resource discovery[16]. This takes care of where the job will be run. When the job will be run is up to the site itself, or specifically the software which handles job queuing.

Chapter 4

jAliEn

4.1 AliEn

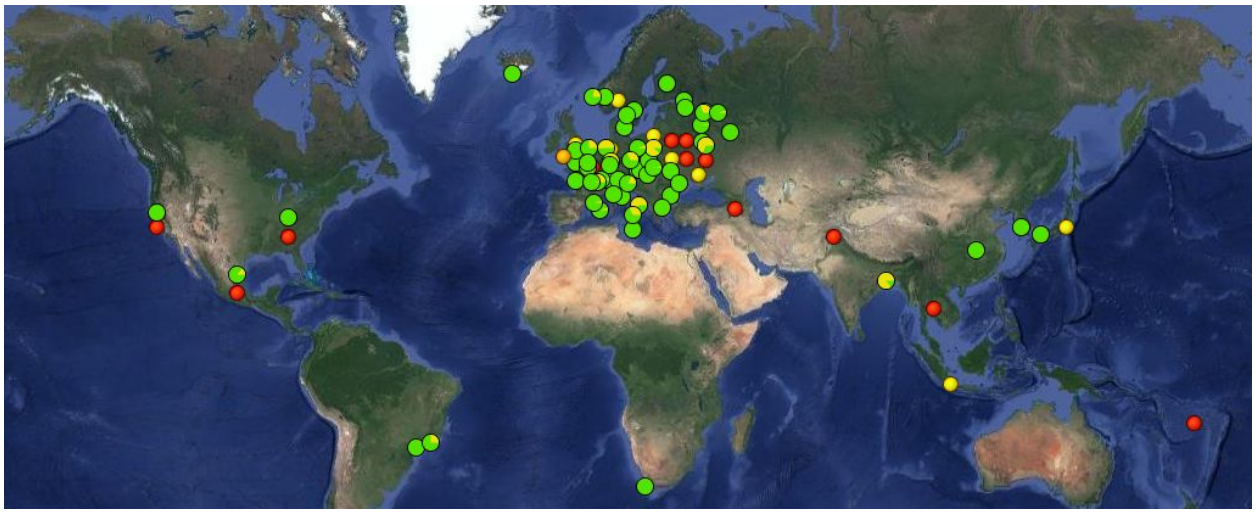

Figure 4.1: Sites map of the AliEn Grid as of 25.09.19

Developed by the Alice Collaboration, AliEn is a Grid framework that has been developed based on Web Services and standard protocols. AliEn also interfaces with other grids. As an example, when interfacing with the European Grid Infrastructure (EGI) the EGI's computing grid is seen as one large AliEn Compute Element (CE), and EGI's storage grid as one large AliEn Storage Element[17].

All AliEn sites fall under one of their 3 tiers, tier 0 - 2. In this system, one or several tier 2 sites would be connected to a tier 1 site, which in turn would be connected to one of the tier 0 sites, the CERN Computer Centre or the Hungarian data centre in Budapest. For instance, the Nordic Data Grid Facility (NDGF) was created to manage Nordic resources and acts as a single entry point for CERN. NDGF also had to create an interface between AliEn and their Grid middleware ARC[18].

The AliEn Grid middleware is composed of several independently working components:

Job Executioner: An AliEn job is defined in the Job Description Language (JDL). The job is eventually matched with a site and a Job Agent (JA) is created which does any necessary preparations like checking that the resource is capable of running the job and cleaning up after it's done. When the JA has started, the job is executed[29].

Storage: The massive amounts of data is stored on the different AliEn sites. Here the tier 0 site distributes the raw data to the tier 1 sites, tier 1 sites then distributes to tier 2 sites, and so on [20].

The file catalogue: For all intents and purposes, the AliEn file catalogue is navigated similarly to a UNIX filesystem from the perspective of the user. Behind the scenes, the AliEn file and metadata catalogue is a database with pointers which provides mapping of files between different storage elements on different sites[29].

MonALISA: AliEn monitoring follows the Monitoring Agents using a Large Integrated Services Architecture (MonALISA) closely. Each AliEn service regularly sends monitoring data to the local MonALISA service running on the site. For a full overview of current AliEn jobs, see the MonALISA Repository for ALICE[21].

As of today (september 2019) AliEn consists of 13 tier 1 sites, and approximately 155 tier 2 sites around the world (see figure 7). Running well over 100 000 jobs at any given time. This number is only expected to increase in the coming future. As a result, Java ALICE Environment (jAliEn) was created to modernize all parts of the distributed system software and fulfill the computing needs of the future.

4.2 Components

jAliEn consists of several components, each with different functions. This section will present each component starting with the central services, called jCentral.

4.2.1 jCentral

jCentral is responsible for the job queue and data management. Functions related to the queue including submissions, splitting jobs into smaller tasks, calculating and controlling job quotas per user, handling the information and parameters of the CPU resources located in different computing centres. It also registers job outputs, status transition, matching and tracing.

On the data management side it decides the file placement and source for all Grid jobs.Taking into consideration the location of the client and the status of the storage and network topology between client and storage servers. All files are kept in a file system like hierarchy within the File Catalogue. For information on the File Catalogue, see section 4.2.3.[38]

4.2.2 jBox

jBox is a service which connects the different sites to the central services (jCentral). It interfaces with jCentral and deals with the local resource management of each site its behalf. It deals with new job submissions and monitors messages from jobs and other services running at the site.[38]

4.2.3 Storage - The File Catalogue

The total storage capacity of the AliEn grid can now be measured in Exabytes (1 EB = 1,000,000 TB). The total capacity is not concentrated in one location, instead it is spread across multiple sites all across Europe. This subsection will describe how files and storage are handled.

The File catalogue in AliEn does not own the files it displays like a normal computer does. It provides the mapping between the Logical File Names (LFN) which is visible to the end user and the Physical File Name (PFN)[29]. The PFN describes the name of the AliEn storage element (SE) and then the path to the local file. The LFN is what's visible to the user, for example */users/username/home* and can be manipulated by moving a file from one directory to another. In order to prevent duplicate file entries (as multiple LFNs can point to one PFN), each PFN is associated to a Globally Unique Identifier (GUID) entry.

4.3 Security

jAliEn uses the X509 Public Key Infrastructure which specifies formats for certificates and a validation algorithm for the certification chain[39]. jAliEn expanded the standard X509 model by introducing "Token Certificates". Unlike proxy certificates discussed in section 3.3.1, a token does not have the same rights as the certificate holder which it was created for. As an example; you can create a token which can operate on just one job on behalf of a user, but it is not authorized to create new jobs or act on other jobs created by the same user[38].

4.4 Installation

Part of this project took place on an independent version of jAliEn installed on a virtual machine running locally at HVL. For a complete description of the installation see Appendix A.

4.4.1 CernVM

A challenge when dealing with a large grid is making sure every computer and virtual machine is running the correct software, and the correct version of that software. A *virtual appliance* is a software image containing a complete system designed to run on a platform. Using a virtual appliance can simplify the set-up process by removing the need for manual installation, as well as configuration and maintenance. This provides the ability to easily create a homogenous environment on a large scale[40].

CernVM is a virtual appliance created for participants of the LHC experiments at CERN[30]. Once it is deployed, instances can be configured to serve a variety of use-cases. If there is a

need for additional software, these can be obtained using the CernVM File System (CernVM-FS).

The CernVM-FS is a software distribution service[43]. In essence, it’s a file system which contains a collection of software used by scientists. You can connect to the file system through the internet and then mount it. Allowing you to access and download the files there.

Chapter 5

The Interactive Python Shell for JAliEn

This chapter will give a comprehensive description of the components and features of the interactive python shell for JAliEn. There will be a general overview of the prototype (5.2), a description of the different libraries used by the application(5.3) before section 5.4 describes how different features were implemented on a technical level. This chapter will start with a brief discussion on the design philosophy of the prototype (5.1).

5.1 Design Philosophy

Event based programming or event-driven architecture is a programming style where the application executes in response to events(se figure 5.1). In the context of this application; an event could refers to a button on the keyboard being pressed. Pressing any of the number or character keys could echo them to the screen and store them in sequence in a string. Pressing the “enter” key could execute the string.

Using events is ideal when the user is in control of the flow of the application during its runtime, i.e when the code interacts directly with the user. An application like the one described in this thesis; which can spend a majority of its runtime waiting for the user to perform an action lends itself very well to being developed using events.

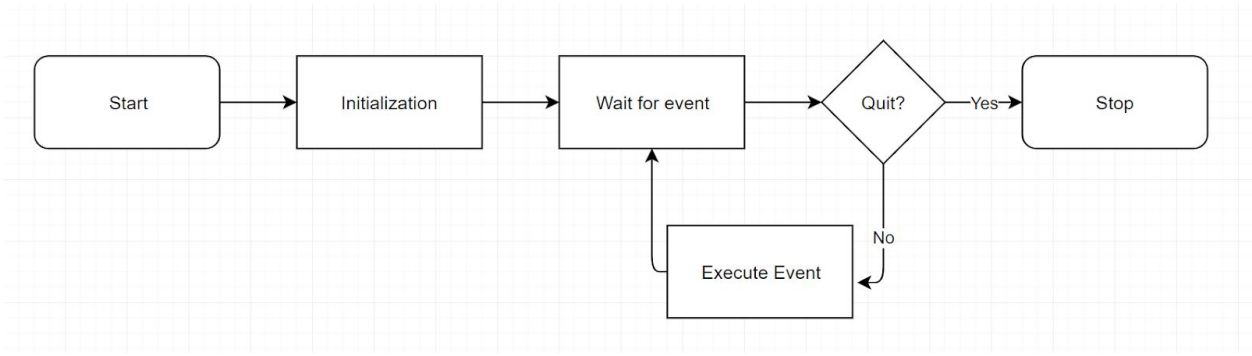

Figure 5.1: Event driven program illustration

Whether or not to use events isn't the question, but to what extent. In other words, what should constitute an event? The application could handle every keypress as an event, that would give the developer a lot of control, but result in quite a bit more code, which would reduce code-readability and maintainability.

When programming with events, it's important to understand the concept of the "main loop". You can see the main loop in figure 5.1. It's where the application waits for an event, then executes it, then waits again.

Event driven programming pros:

- Control - Event based programming allows the developer to perform an action every time the user presses a key.
- Performance - Event based applications use very little resources when no event is being triggered
- Common - Event based programming is commonly used, as a result there are several well documented libraries to choose from.

Due in part to the event library chosen (cmd2) an event (i.e what causes the application to move out of the main loop) in the prototype is the user pressing the "enter" key. All other key presses are handled by the readline, which only returns when enter is pressed.

5.2 Introduction and Overview

The section aims to give an overview of the applications features, including going into some technical details. This section will describe the what and not the how. How these features were implemented is described in section 5.4. A visual overview is shown in figure 5.2[2].

[2] The figure simplifies parts of the application by packaging some steps together in initialization, parsing and executing input and re-establishing the connection.

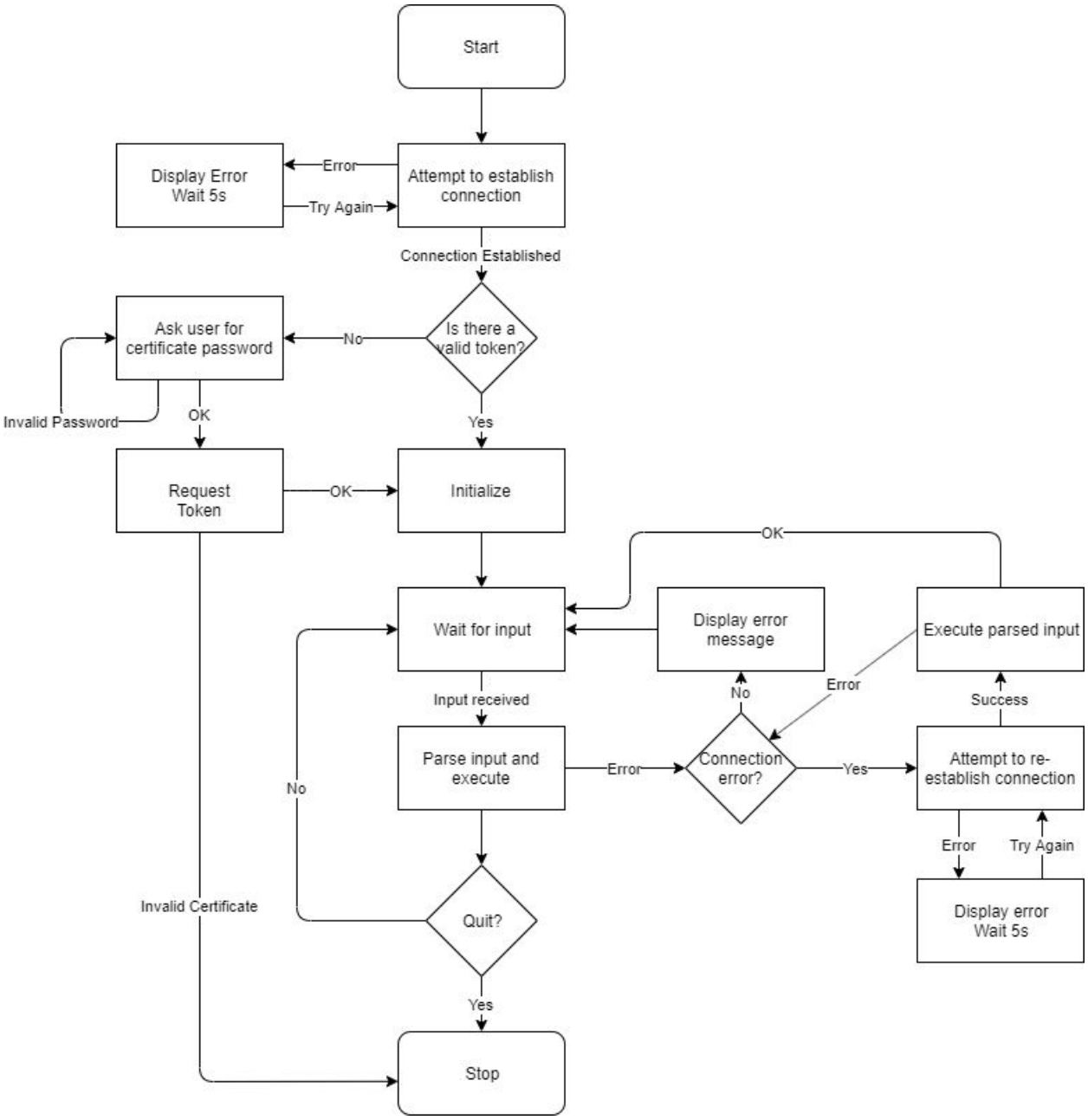

Figure 5.2: Application runtime overview

The interactive python shell for jAliEn attempts to emulate the unix shell wherever possible. As far as interactive CLIs go, the unix shell is an established standard. More importantly it's something the users are familiar with, as such it will flatten the learning curve. Before the components of the python shell is described in detail, this section will describe the unix terminal and bash which it's trying to emulate and how the python shell compares.

In the world of computing, a shell is a UI for accessing the operating systems' services. The Bourne-again shell, or bash, is a Unix shell and a command language (a language for job control in computing). On top of being able to execute commands from an interface, it can also execute them from a file; this is called a shell script. Features such as auto completion, help and history are all found in bash.

The unix terminal is, put simply; an interface in which you can type and execute text based commands i.e a CLI. The "commands" refer to executables located on your computer or built in commands in the shell. The location of the executables are set using the variable PATH. If you type in a command (not including the location of the command) it will search through the built in commands first, then the directories listed in PATH, and execute it if it finds it.

The python shell works quite differently. The commands don't refer to executables located on your computer but methods in the script. The *ls* command refers to the method do_ls, the *cd* command refers to the method do_cd and so on. The commands which exists in jshell as well as in bash have two variations, one if executed on the grid and one if executed locally. Using the command *switch* the user can change whether they want the commands to be run locally or be sent to the central services. Alternatively preceding any command with an exclamation mark will cause it to be run locally.

This difference will go unnoticed by most users. It is only if a user wants to add a command where this might be a hindrance. As there is no PATH variable the user has to add a new method to the script if they wish to add a new command.

Autocompletion is a key feature of bash. By typing the beginning of a command/path and then pressing the tab key, you either get all possible completions if there are more than one, or it auto completes the command if there is only one possible completion(see figure 5.3). In this context there are two parts to auto completion; **readline** and bash's own built in **complete**.

```
$ git<tab><tab>
git                git-receive-pack    git-upload-archive
gitk               git-shell           git-upload-pack
$ git-s<tab>
$ git-shell
```

Figure 5.3: Tab completion example[31].

Readline manages the command line editing, and is responsible for handling the tab press. When tab is pressed readline calls back to the bash completer which returns with all possible completions. The completer defines the completion mechanism for commands. If no command is found to match the input, it attempts to compare it to paths/file names instead.

Here, the python shell implements a very similar approach, taking advantage of readline while creating its own completer. For a detailed rundown of how tab completion was implemented see section 5.1.1 and 5.3.1.

The prototype is an interactive command line interface, though it doesn't actually handle any of the commands. When the user inputs a valid command, it is sent to the central services (see section 4.2.1) which processes the command and replies with a response. This response is then parsed and relevant data is displayed. The file sent to the central services via WebSocket is a simple json with two parameters; "command" and "argument".

The application exist only in the form of a python script which is executed from the terminal. First, the websocket connection is established and if successful the settings will be set before the program enters the main loop where it stays until the application is closed.

5.2.1 jalien_py

The python shell consists of two modules. The interactive shell described in this thesis uses jalien_py as a library. It is responsible for, among other things, handling much of the back-end; specifically the websocket connection and xrootd[3].

Figure 5.4 shows how the scripts interact[4]. Everything on the right side marked in red is handled by jAliEn_py and everything else by the interactive shell. After an input is received from the user; it is parsed by the interactive pyshell. If the input is a command which requires communication with the central services it is sent to jalien_py to be packaged into a json which is sent to the websocket endpoint. The application then waits for a response and once one is received, jalien_py unpackages the json, prints any relevant results before the application waits for a new input.

[3] Xrootd is a “file access system” used for file management (copy, download, etc)[28].
[4] Assuming no errors occur.

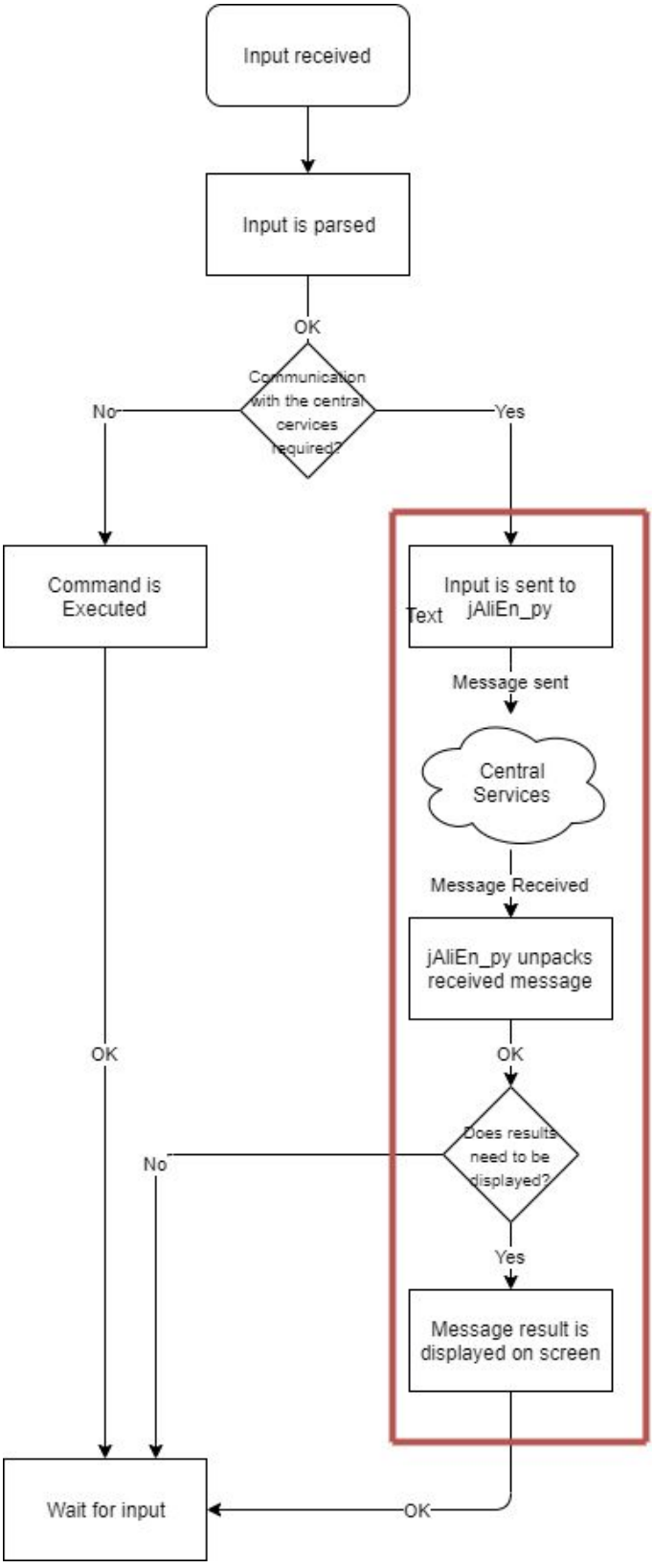

Figure 5.4: Script Interaction Illustration (jalien_py in red)

5.2.2 Additional features

The prototype allows the user to run commands locally on the computer as if they were using the regular unix terminal. As a result, users can write scripts which takes advantage of commands found in bash.

As an example, take the grep command. It searches the input file(s) for a match to a given argument and echoes them to standard output (by default). Implementing this functionality on the prototype to work locally and on the grid would be a big undertaking. What the application can do instead is use the grep found in bash.

Let's say a user wanted to look for lines matching “ABC” in the file “test.txt” on the grid. The application can download the file, run the local *grep* command on the file with the arguments given by the user and then delete the file. From the user’s point of view, this would be no different than having it executed by the central services and having the results sent back.

This can be taken even further. The files sent between the client and server are all json files. Normally, these files are unpacked automatically and the user only have access to the parsed results in the form of a string. By giving the user access to the raw json data they can manipulate the results as they see fit using a json processor like jq[5]. Jq gives you the ability to search through a json result in a similar way to grep and gives you access to information about the message received from the central services that would otherwise be deleted in the unpacking process.

The prototype supports all commands currently implemented in jAliEn. This includes commands acting on the grid’s file system (ls, cd, cp) and commands for interacting with jobs (submit, kill). For a full list of supported commands and what they do see appendix B.

[5] https://stedolan.github.io/jq/

5.3 Libraries

Investigating and testing different libraries designed to make the development of CLIs easier was a big part of development. This section will give a description of the libraries which makes the application run.

5.3.1 cmd2

The first library tested for use in the application was cmd[23]. Cmd is part of the Python Standard Library and provides a simple framework for writing line-oriented command interpreters. The library offers many useful tools, however comes with one major drawback; it has control over the main loop. Since the framework offers a relatively small amount of useful features that was enough not to consider it a viable option.

cmd2 is a python package for building powerful CLI programs[24]. It extends the Python Standard Library's cmd package. Including backwards compatibility with cmd, it makes use of other libraries to expand the framework's functionalities. The most impactful libraries being argparse and readline.

Argparse, much like cmd and cmd2 is a module designed to make it easy to write your own CLI[25]. It allows you to define commands, sub-commands and arguments(see figure 5.5) and in doing so the library will handle the parsing for you. Cmd2 makes use of argparse's functionalities to handle argument processing and the creation of commands. A developer don't need to be aware of this in simple use cases, as cmd2 will handle all aspects of argparse for you.

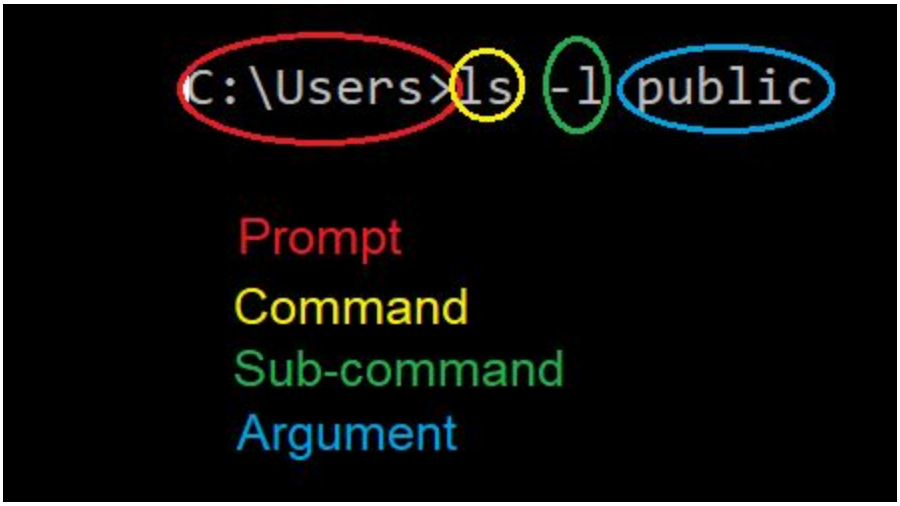

Figure 5.5: The structure and terminology of a terminal input

Readline allows the developer access to the raw input as it is typed in[26][27]. Allowing you to read, write and alter it. This is crucial when developing an interactive CLI. Cmd2 use readline indirectly to enable tab completion and history.

Cmd2 has built in support for a variety of settings. Settings include the ability to set a history file, decide the initial prompt(see figure 5.5), whether to show full error stack and more. It also allows the developer to create their own settings.

Cmd2 still has the same major drawback cmd has; it has control of the main loop. For a majority of use-cases this isn't an issue, however as described later this will impact the development and usability of this application. With the comprehensive list of features this library contains, it was something worth developing around.

Figure 5.6 gives a simplified view of cmd2's main loop. Pre-loop is run at the start of the application's lifecycle. It initializes the settings if you've edited any. After the pre-loop the application will wait for an input. Once an input has been given it will run the pre-cmd method and hooks. The hooks are any methods written by the developer which run before every command. These methods have access to the input, as a result certain hooks can be made to run only before certain commands.

In cmd2; a command is a method with the same name pre-faced with “do_”. As an example; the *ls* command is the method named “do_ls”. This method is run after pre-cmd. The post-cmd method and hooks are run after the cmd before the application waits for another input. Once the user quits the program the post-loop is run. The pre- and post-loop/cmd methods are empty to begin with and can be used by the developer if necessary.

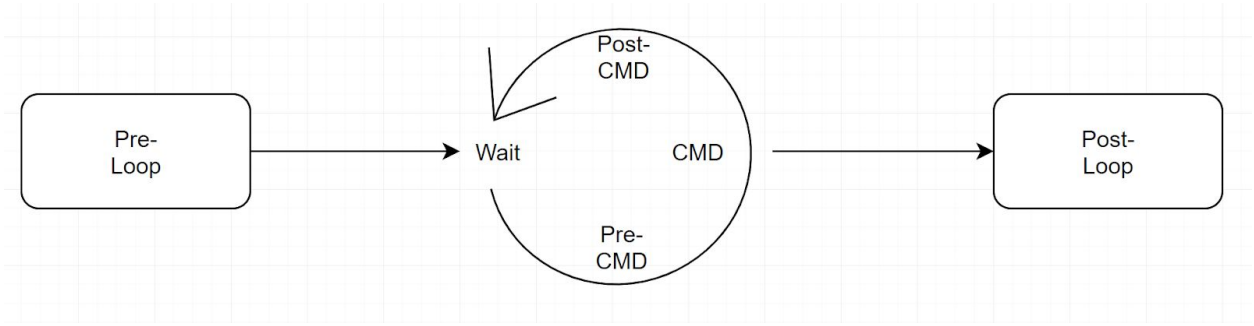

Figure 5.6: A simplified illustration of the main loop

To summarize, cmd2 comes pre-packaged with the following features relevant for this project:

- **Commands** - Creating new commands is as easy as declaring a new method
- **History** - cmd2 offers readline history which can be searched, edited and stored.
- **Tab Completion** - cmd2 come pre-packaged with tab completion for commands and support for path completion.
- **Aliases** - Creating aliases for existing commands. For example *quit* has an alias *q*.

5.3.2 AsyncIO

AsyncIO is a library used to write concurrent code using the async/await syntax[32]. As the name suggests, it allows the developer to run code asynchronously. As an example, after the shell has sent a command to the central services it **awaits** a response. Using AsyncIO, the application could run multiple tasks at the same time. Since that isn’t necessary in this context, the application is instead told to run the WebSocket task until it is complete before moving on.

5.3.3 WebSockets

The client is supposed to connect to the central services' websocket endpoint. As a result the prototype uses the python websockets library[6]. Websockets is a library for building WebSocket servers and clients for python which provides low level APIs for WebSocket.[35]

5.4 Components

This section will describe how the features of the application were implemented and discuss relevant alternatives .

5.4.1 Tab Completion

As discussed earlier, cmd2 comes with support for tab completion for commands and local path. It does so via a combination of the argparse and readline modules. An issue arises when you attempt to implement tab completion on the grid file system. Having tab completion on the grid is especially important due to the long auto generated file names (see figure 5.7). The application is restricted, as it is only able to communicate with the central services by sending commands and receiving the results back. We don't have any direct access to the grid's file system like we do with the user's own machine.

```
Run282683_282683_v2_s0.root
Run282700_282700_v2_s0.root
Run282702_282702_v2_s0.root
Run282703_282703_v2_s0.root
Run282704_282704_v2_s0.root
Run282708_282708_v2_s0.root
Run282710_282710_v2_s0.root
jsh: /alice/data/2017/OCDB/TPC/Calib/Temperature/ >
```

Figure 5.7: Example of file names found on the grid.

Cmd2 contains support for creating your own argparser object for a command(if an argparser is not manually created, cmd2 will create one for you). The argparser you have created can then be attributed to a command by using the "with_argparser" decorator on that command's

[6] https://pypi.org/project/websockets/

corresponding method. It's the information about a command stored in the argparser object which is used to decide the suggestions for auto completion by readline.

Attempt 1: When the websocket connection is established, and every time the user presses the "tab" key; an *ls* command is sent to the central services. The results of the ls request is used to create a new argparser for the relevant commands.

This is where the drawback of cmd2 comes into play. Since cmd2 has control over the main loop the developer lose the ability to affect the application when it's waiting for input. As a result, the developer cannot perform any action when the user presses the "tab" key.

Attempt 2: Retake control of the main loop by executing the necessary parts of cmd2 manually. The Pre- and post-loop methods, as well as the pre- and post-cmd plus hooks methods can all be executed manually. Though not one of cmd2's main focuses, it does have support for integration with other event loops in this manner. This would effectively create an inner loop which we have control over, and an outer loop(see figure 5.8).

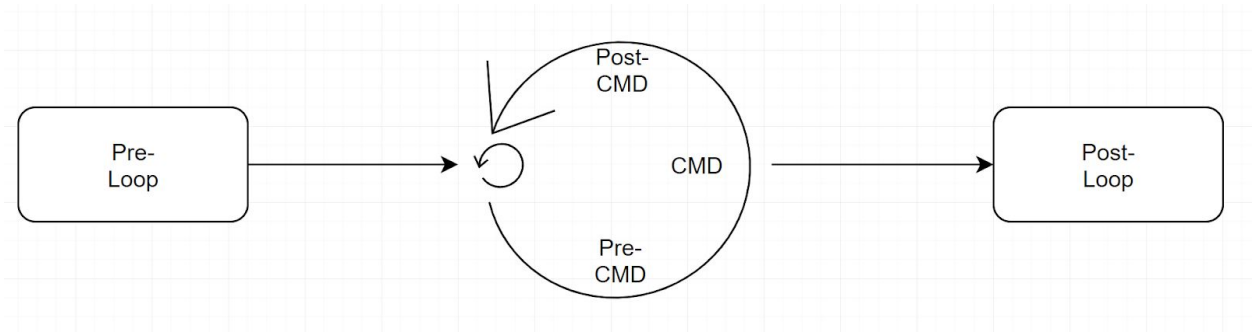

Figure 5.8: An illustration of solution two.

By utilising this solution, the application lose many of its key features and to regain them the new main loop would need to contain all the functionalities of the old one. This heavily impacts maintainability, as a developer would need to create their own patch any time cmd2 releases a new patch.

Attempt 3: Static tab completion. Due to a lack of control over the main loop, an ls request is sent any time the user changes directory. The argparse for the relevant commands is updated to include all of the files and directories in the current directory only. If a user wants to autocomplete file or directory names in another directory, they first have to change to that directory.

It does reduce the user-friendliness of the overall program but due to the discussed constraints, it's the solution implemented in the prototype .

The best solution found allows for dynamic tab completion on the grid. Cmd2 has limited support for integration with other event loops, meaning it supports it but you lose many key features in the process. The ideal solution found involves replicating the necessary features from cmd2, removing the prototype's dependency on it and gaining control of the main loop as a result.

Readline has a variety of settings a user/developer can edit. Including key bindings and changing certain values. One of which is the ability to decide which completer readline should use to retrieve suggestions. The completer can be as simple as an algorithm which takes a string as input and searches a list of words for matches to the input, which it returns(see table 5.1).

```
list =   ["Some", "Words"]
def completer_example(input):
       //compare input to each word in list
       //return matches
```

Table 5.1: Simple completer example

By gaining control over the main loop, the application would configure the completer in such a way so that pressing tab sends an ls request to the central services(see table 5.2). The result of this ls request would be used to update the list of defined arguments in argparse. Essentially allowing the implementation of the solution presented in attempt 1.

```
def completer_example(input):
      //send ls request to central services
      //update argparse with new arguments from ls request
      //create new completer from the updated argparse object
      //perform completion
     //return matches
```

Table 5.2: Feature complete completer example

After sending an ls request to the central services and waiting for a response, the application is essentially sleeping. If this action takes too long, a user can perceive this as the client freezing up. The time it takes for any round-trip communication will depend on the user's connection to the internet, and their distance to the central services. During testing of the prototype, the delay never reached one second (excluding cases where the connection broke and had to be re-established).

5.4.2 Scripting

The application supports both running scripts from a file, and writing multi-line scripts in the terminal window. The *py* command will enter an interactive python session with full access to the application's code. As discussed earlier; a command (say *ls*) is just the method "do_ls". As such the user can create scripts using the methods directly. Changes made by the script will persist for the lifetime of the application; as an example any variables declared by the user will be remembered by the application until it is closed.

5.4.3 Robustness

After a connection has been established and the user successfully logs in, they will be directed to their home directory. If the connection is broken and then re-established, the user will by default be directed back to the home directory no matter where they were when the connection broke. To solve this issue, the current directory is stored as a string locally. This is also done so it can be used as part of the prompt(see figure 5.3).

Due to the architecture of the application, a connection can only be re-established when the user attempts to send a request to the central services. If it detects a broken connection at this point, it will attempt to reconnect. If the connection is successful, a “change directory” command will be sent to the central services before the command requested by the user is sent. As a result, a broken connection which can be re-established will only result in a second of lag from the user’s point of view. In the event where a connection cannot be established, an appropriate error message will be displayed and the application will attempt to reconnect every 5 seconds.

Every input is parsed and checked for errors before being sent on to the central services. Every command has a known number of valid arguments. Knowledge of what those arguments are is not always possible. As a result, invalid arguments can be sent to the central services (attempting to remove a file which doesn’t exist) however in such cases the central services will simply respond with an appropriate error. Knowing the number of arguments each command has and recognizing the difference between an argument (which can vary) and a subcommand (which is static) results in the client never sending an invalid command.

5.4.4 Maintainability

The script is both easy to read and add too. Each of the available commands exists as its own method, therefore a developer should not have to alter any existing code when adding support for commands in the future.

5.5 Installation and Requirements

Installation of the software is done via CernVM(see section 4.2.1). After connecting to CernVM, the user should simply navigate to the location of the application and extract it. In order to log on you will need a valid certificate. For instructions on how to get one, visit the AliEn website[7]. In order for the application to have access to the certificate, the user has to set the environment variables *X509_USER_CERT* to be the path to the user certificate and *X509_USER_KEY* to be the path to the private key. If the environment variables are not set, the application will assume they are located in the *~/.globus* folder.

[7] https://alien.web.cern.ch/content/vo/alice/getcertificate

Chapter 6

Evaluation

Chapter 1 detailed the features which constitutes an interactive shell. Having investigated viable options and developed a functioning prototype, this chapter aims to compare the resulting application with the criteria set in Chapter 1 and determine whether or not the prototype could be a viable replacement or alternative to jShell.

6.1 Fulfilling the Criteria

Section 1.3.1 outlined a number of requirements for a viable solution. Section 5.3 detailed how these solutions were implemented. Section 6.1.2 will discuss whether or not these requirements were met but first there will be a brief recap of the criteria and how they function in the prototype.

6.1.1 Criteria Recap

- Tab Completion
- The prototype has limited support for tab completion. It can complete commands, sub-commands and arguments (including file-path) both locally and on the grid. However due to technical limitations described in section 5.3.1 it does not work outside or the current directory on the grid.
- Scripting
- The prototype has full support for scripting. By giving the user access to the script's code they can create scripts by either typing it directly in the terminal or load it from a file. This code has to be written in python.
- Robustness
- The prototype will handle errors caused by both user input or connection. All input is parsed before it is executed, as such errors/mistakes in the input will be caught before it is sent further down "the pipeline" where such errors could cause a problem.

The connection to the central services is kept open during the application's runtime (or until it is timed out) so to avoid unnecessary handshakes. If the connection is broken it will attempt to reconnect, get the user back to the directory they were at and send the command to the central services with no intervention required by the user.

- Light
- The prototype consists only of python code with no graphical assets and seeks to utilize existing terminal features wherever possible (instead of unnecessarily replicating them). There are only a handful of libraries and no large dependencies. The final prototype is 19KB in size.
- Maintainability
- The script was designed to be easily readable and follows basic design principles (low coupling high cohesion). Each separate command exists in the form of a method, as such new ones can be added and old ones removed without issue.
- User Friendly
- The prototype tries to emulate the unix terminal/bash wherever possible. What this means is that when a new user uses the application for the first time they are met with a familiar UI with familiar functionalities.

  Installation will be done via CernVM, which again is familiar to the majority of users and the additional dependencies (the certificates) are by default located in the same directories as before.

6.1.2 Satisfying Functional Criteria

In section 5.2, an existing library which was found that could help satisfy a number of set criteria. By utilizing components of cmd2 a baseline prototype which fulfilled several of the criteria was developed.

The most baseline functionality/criteria of the application is the ability to display a CLI, take some user input, send it to the central services and display the returning results. The prototype supports all commands found in JAliEn and will parse received input and display the appropriate result. As such, this criteria is considered satisfied.

Due to the long auto generated file names found on the grid having functional tab completion was deemed necessary. Several possible solutions were explored and eventually due to technical limitations described in section 5.3.1 a compromise was found which allowed for tab completion on the grid without having to scrap the entire prototype and start over. Though tab completion doesn't work flawlessly as described in the section above, this criteria is considered satisfied.

Scripting works as originally intended when the criteria was set. Any and all commands supported by JAliEn can be used in user-made scripts. Python is well known object oriented language and any features found in the python language can be utilized by the scripts. The application can load these scripts from a file or alternatively they can be written directly into the terminal. As a result, this criteria is considered satisfied.

6.2 Determining Viability

As mentioned in section 4.4.1; one of the challenges when dealing with a large grid is making sure every system runs the correct software and the correct version of that software. The existing client (jShell) is written in Java, which requires the correct version of Java to be installed on any computer wanting to run it.

Both Java and Python comes pre installed on most Linux distributions. The problem with Java is backwards and forwards compatibility. Oracle releases a new version of java twice a year. If jShell is updated and then compiled with the newest version of Java, it won't run on older Java versions. This is not the case with Python. All versions of Python 3 are backwards compatible with previous versions of Python 3. As a result, a client written in python alleviates the issue of having to take compatibility into consideration when updating the client.

Java is a commercial product owned by Oracle. Oracle could at any time decide to implement a license fee or make changes detrimental for certain applications. Python on the other hand is open-source and maintained by The Python Software Foundation, a non-profit organization. As a result, a client written in Python is preferred.

Due to the advantages of having a client in Python over Java, and because the functional criteria were satisfied, the new client is considered a viable replacement/alternativ of jShell.

Chapter 7

Conclusion

Grid technologies provide scientists all across the world the computing resources needed to make new discoveries and further our understanding of the universe and everything in it. JAliEn is a set of interconnected components designed to connect computing centres together into one large homogenous entity. A client needs to be able to interact and access the tools made available by jAliEn and due to faults in the old client jShell, the need for a new client written in python arose.

Based on what constitutes an interactive CLI, criteria was set and the development of a solution began. Bash is a commonly used CLI amongst JAliEn's user base and is an established CLI standard, as a result the prototype client took a lot of its inspiration from the bash shell and attempts to recreate many of its features. By utilising components found within another python library (cmd2), many of the criteria were met. To fully satisfy the criteria, further development was undertaken to enhance the capabilities of the prototype.

By utilising readline in a similar manner to bash, and continually updating the list of possible completions, tab completion on files in the grid was made possible. Giving the users access to the scripts' code base allows them to create their own scripts using the commands supported by jAliEn. Making each command independent of each other and adhering to common programming principles resulted in a script which is easy to read and maintain.

To determine the viability of the solution, the functionalities of the prototype was tested on the criteria set in section 1.3 and compared to the existing client; jShell. For robustness, every input is parsed and checked for errors before being sent on to the central services and a (temporary) loss of connection will automatically be corrected by the application.

Reviewing the functionalities achieved, the results gained from testing the criteria and the comparison to the existing client, it can be concluded that the solution is both a viable alternative and replacement of jShell and it's viable in correlation with the set criteria.

7.1 Further work

Though the prototype attained the set criteria as discussed earlier in this chapter, improvements can still be made. By removing the dependency on cmd2 from the application and instead replicate the necessary features from the library it will take control over the main loop which could allow for full tab completion. Additionally, documentation could be written so that the users know of the application’s functionalities.

Appendices

Appendix A

JAliEn Installation

A.1 Prerequisites

In order to install JAliEn, the VM needs the following: **git**, the **xrootd client**, the **java openjdk version 8** and of course the **JAliEn client**. To install the prerequisites and compile JAliEn type the following commands:

- yum install git
- yum install xrootd-client
- yum install java-1.8.0-openjdk-devel
- git clone https://gitlab.cern.ch/jalien/jalien.git
- cd jalien
- ./compile.sh all

A.2 Setup

- First, navigate to the home directory, and make the directories .globus and .j.
    - cd ~/
    - mkdir .globus
    - mkdir .j
- Copy the directories ~/jalien/config and ~/jalien/installation/trusts/trusts to the .j folder.
    - cp -rf ~/jalien/config .j
    - cp -rf ~/jalien/installation/trusts/trusts .j
- Remember to add the certificate of the CA(cacert.pem) to .j/trusts
    - cd .globus
- Move hostcert.pem and hostkey.pem to .globus
- Hostcert/key and usercert/key has to be identical. You can use symlinks
    - ln -s hostcert.pem usercert.pem
    - ln -s hostkey.pem userkey.pem
- JAliEn requires a p12 CA certificate to load the necessary libraries

- It doesn’t matter which CA, so you can make a self-signed one
    - openssl genrsa -des3 -out rootCA.key 4096
    - openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.crt
    - openssl pkcs12 -export -out alien.p12 -inkey rootCA.key -in rootCA.crt
    - cd ~/.j/config
- The file “password.properties” has to exist in config, or else it is assumed that JAliEn is just a load balancer
    - echo "password = DB_PASSWORD_HERE" > password.properties
- Edit the properties files so that it matches the rest of the setup (LDAP and DB)
- If you’re using a p12 certificate with a password, you have to add “ca.password” to “config.properties”.

A.3 Execution

- Navigate to the jalien directory
    - cd ~/jalien
- Run jCentral
    - ./runJCentral.sh
- Start jBox
    - ./jalien login
- Start jShell
    - ./jalien

Appendix B

List of Commands

Below is a list of all commands supported by the interactive python shell for jAliEn.

- echo - prints argument to default output, can be redirected to file
- quit - Exits the application
- cd - change directory
- ls - Lists all entities in the current directory
    - a - shows hidden files
    - l - shows long listing
    - S - sorts listing by names
- less - reads content in file
- switch - Change between acting on the local file system and the grid
- get -
- ls_csd - Runs the ls command in Cassandra
- cat - Reads a file and writes it to output
- cat_csd - Runs the cat command in Cassandra
- whereis - Locates source/binary and manuals sections for specified files
- whereis_csd - Runs the whereis command in Cassandra
- cp - Copies file
- cp_csd - Runs the cp command in Cassandra
- time -
- mkdir - Creates Directory
- mkdir_csd - Runs the mkdir command in Cassandra
- find - Locates matching filenames
- find_csd - Runs the find command in Cassandra
- listFilesFromCollection -
- submit - Submit Job
- motd - Message of the day
- access -
- commit -

- packages - List available packages
- pwd - Prints current directory
- ps - Reports information on running processes
- rmdir - Remove directory
- rm - remove file
- mv - Move file
- mv_csd - Runs the mv command in Cassandra
- masterjob -
- user - Change role of specified user
- touch - Create file
- touch_csd - Runs the touch command in Cassandra
- type -
- kill - Kill process
- lfn2guid - Prints GUID for given LFN
- guid2lfn - Prints LFN for given GUID
- w - Get list of active/waiting jobs
- uptime - Get list of running/waiting jobs and active users
- chown - Changes and owner or group for a file
- deleteMirror - Removes a replica of a file from the catalogue
- df - Show free disk space
- du - Show disk space usage of current directory
- fquota - Displays information about File Quotas
- jquota - Displays information about Job Quotas
- listSEDistance - Returns the closest working SE for a particular site
- listTransfer - Returns all the transfers that are waiting in the system
- md5sum - Returns md5 checksum of given filename of GUID
- mirror - Mirror copies a file into another SE
- resubmit - Resubmits a job or a group of jobs by IDs
- top - Display and update information about running processes
- groups - Show the groups current user is a member of
- token -
- uuid - Returns info about given LFN
- stat -

- listSEs - Print all (or a subset) of the defined SEs with their details
- xrdstat -
- whois - Usage whois [account name]

References

1. ALICE. Available From: https://home.cern/science/experiments/alice [26 August 2019]
2. I. Foster, C. Kesselman, S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3), 2001. Available at https://www.globus.org/sites/default/files/WhatIsTheGrid.pdf
3. A. G. Grigoras, C. Grigoras, M. M. Pedreira, P. Saiz, S. Schreiner. JAliEn - A new interface between the AliEn jobs and the central services. Journal of Physics: Conference Series 523 (2014). Available at http://iopscience.iop.org/article/10.1088/1742-6596/523/1/012010/pdf
4. About HTML5 WebSocket. Available From: https://www.websocket.org/aboutwebsocket.html [28 August 2019]
5. Our History. Available from: https://home.cern/about/who-we-are/our-history [10 September 2019]
6. Member States. Available from: https://home.cern/about/who-we-are/our-governance/member-states [10 September 2019]
7. Our People. Available from: https://home.cern/about/who-we-are/our-people [10 September 2019]
8. The Large Hadron Collider. Available from: https://home.cern/science/accelerators/large-hadron-collider [10 September 2019]
9. ALICE. Available from: https://home.cern/science/experiments/alice [10 September 2019]
10. The Large Hadron Collider: Conceptual design - LHC Study Group (Pettersson, Thomas Sven et al.) CERN-AC-95-05-LHC. Available from: http://inspirehep.net/record/402898/files/cm-p00047618.pdf
11. Palomo, L.. (2017). The ALICE experiment upgrades for LHC Run 3 and beyond: contributions from mexican groups. Journal of Physics: Conference Series. 912. 012023. 10.1088/1742-6596/912/1/012023.

12. I. Foster, C. Kesselman. The History of the Grid. High Performance Computing: From Grids and Clouds to Exascale, Advances in Parallel Computing Series., vol. Vol. 20, pp. 3-30, 2011, iOS Press. Available at http://www.ianfoster.org/wordpress/wp-content/uploads/2014/01/History-of-the-Grid-numbered.pdf
13. R. Kumar, A. K. Shukla, R. Shukla. Analysis of Grid Computing Architecture & Its Real-Time Application. International Journal of Innovative Research in Science, Engineering and Technology Volume 4, Issue 9, September 2015. Available at https://www.ijirset.com/upload/2015/september/140_Analysis.pdf
14. Public Key Infrastructure. Available from: https://docs.microsoft.com/nb-no/windows/win32/seccertenroll/public-key-infrastructure?redirectedfrom=MSDN [16 September 2019]
15. User How-To - Jobs. Available from: https://alien.web.cern.ch/content/documentation/howto/user/jobs [17 September 2019]
16. Junwei Cao, D. P. Spooner, S. A. Jarvis, S. Saini and G. R. Nudd, "Agent-based grid load balancing using performance-driven task scheduling," Proceedings International Parallel and Distributed Processing Symposium, Nice, France, 2003, pp. 10 pp.-. doi: 10.1109/IPDPS.2003.1213139. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1213139&isnumber=27277
17. S. Bagnasco, P. Cerello, R Barbera, P. Buncic, F. Carminati, P. Saiz. AliEn - EDG Interoperability in ALICE. Computing in High Energy and Nuclear Physics, 24-28 March 2003, La Jolla, California. Available at http://www.slac.stanford.edu/econf/C0303241/proc/papers/TUCP005.PDF
18. P. Gros, A. R. Gregersen, C. Grigoras, J. Lindemann, P. Saiz, A. Zarochentsev. Interoperating AliEn and ARC for a distributed Tier1 in the Nordic countries. Available at https://www.researchgate.net/publication/228678316_Interoperating_AliEn_and_ARC_for_a_Distributed_Tier1_in_the_Nordic_Countries
19. S Bagnasco, L Betev, P Buncic, F Carminati, C Cirstoiu, C Grigoras, A Hayrapetyan, A Harutyunyan, A J Peters and P Saiz. AliEn: ALICE environment on the GRID. Available from: https://iopscience.iop.org/article/10.1088/1742-6596/119/6/062012
20. home.cern. The Grid: A system of tiers. Available at https://home.cern/science/computing/grid-system-tiers [24 September 2019]

21. alien.web.cern.ch. ALICE Grid monitoring with MonALICA. Available at https://alien.web.cern.ch/content/documentation/howto/voadmin/monalisa [25 September 2019]
22. Ku-Mahamud, Ku & Jamal Abdul Nasir, Husna. (2010). Ant Colony Algorithm for Job Scheduling in Grid Computing. Asia International Conference on Modelling & Simulation. 40-45. 10.1109/AMS.2010.21. Available from: https://ieeexplore.ieee.org/document/5489677
23. Python Software Foundation. cmd — Support for line-oriented command interpreters. Available from https://docs.python.org/3/library/cmd.html
24. Devlin C. & Leonhardt T. (2019). Cmd2 Documentation Release 0.9. Available From: https://buildmedia.readthedocs.org/media/pdf/cmd2/latest/cmd2.pdf
25. Python Software Foundation. argparse — Parser for command-line options, arguments and sub-commands. Available from https://docs.python.org/3/library/argparse.html
26. Ramey C. The GNU Readline Library (7/1/2019) Available from https://tiswww.case.edu/php/chet/readline/rltop.html.
27. Python Software Foundation. readline — GNU readline interface. Available from https://docs.python.org/3/library/readline.html
28. Dorigo A, Elmer P, Furano F, Hanushevsky A (2005) XRootD - A highly scalable architecture for data access. WSEAS Transactionson Computers, pp 348–353 Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.9281&rep=rep1&type=pdf
29. S Bagnasco, L Betev, P Buncic, F Carminati, C Cirstoiu, C Grigoras, A Hayrapetyan, A Harutyunyan, A J Peters and P Saiz. AliEn: ALICE environment on the GRID. Available from: https://iopscience.iop.org/article/10.1088/1742-6596/119/6/062012
30. Buncic P., Sanchez C. A., Blomer J., Franco L., Harutyunian A., Mato P & Yao Y. CernVM – a virtual software appliance for LHC applications. Journal of Physics: Conference Series, Volume 219, Part 4. Available from: https://iopscience.iop.org/article/10.1088/1742-6596/219/4/042003 [16 November 2019]
31. Creating a bash completion script. Available from: https://iridakos.com/tutorials/2018/03/01/bash-programmable-completion-tutorial.html [13 October 2019]
32. asyncio — Asynchronous I/O. Available from: https://docs.python.org/3/library/asyncio.html [15 October 2019]

33. About HTML5 WebSocket. Available from: https://www.websocket.org/aboutwebsocket.html [16 October 2019]
34. Fette I., Melnikov A. (December 2011). The WebSocket Protocol. Available from: https://www.rfc-editor.org/info/rfc6455
35. Websockets 8.0.2. Available from: https://pypi.org/project/websockets/ [23 October 2019]
36. Gattoll M. (January 10, 2015). Bourne-again Shell. Available from: https://www.markus-gattol.name/ws/bash.html
37. Danesh A & Jang M. (February 2006). Mastering Linux. John Wiley & Sons, Inc. p. 363. ISBN 978-0-7821-5277-7.
38. Martinez P. M., Grigoras C., Yurchenko V. JAliEn: The new ALICE high-performance and high-scalability Grid framework. (September 17 2019). Available from: https://www.epj-conferences.org/articles/epjconf/abs/2019/19/epjconf_chep2018_03037/epjconf_chep2018_03037.html
39. Cooper M., Hesse P. et al., X.509 Public Key Infrastructure (2005). Available from: https://tools.ietf.org/html/rfc4158
40. VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist (whitepaper). https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf
41. Macdonald I. Working more productively with bash 2.x/3.x. Available from: http://www.caliban.org/bash/index.shtml [5 November 2019]
42. Barry Wilkinson. *Grid Computing: Techniques and Applications*. Chapman and Hall, 2009.
43. CernVM File System (CernVM-FS). Available from: https://cernvm.cern.ch/portal/filesystem [16 November 2019]